А. П. Стасенко

ГЕНЕРАЦИЯ ИСПОЛНЯЕМЫХ ТЕСТОВ ДЛЯ КОМПИЛЯТОРА

ВВЕДЕНИЕ

Данная статья посвящена описанию генератора тестов, пригодного для автоматического тестирования компилятора языка программирования и выявления в нём ошибок, связанных с синтаксисом и семантикой языка программирования [1]. Автоматическая генерация тестов является важной вспомогательной частью тестирования компилятора, так как тесты, разработанные вручную, не могут покрыть все возможные комбинации использования конструкций языка. Для автоматизации тестирования компиляторов были разработаны многочисленные подходы [2].

Генерация синтаксически правильных тестов не представляет затруднений и обычно осуществляется с помощью легко создаваемой контекстно-свободной грамматики. Однако генерация компилируемой (статически семантически правильной) программы намного сложнее, так как в ней требуется обеспечить выполнение семантических условий, таких как использование только определённых переменных и т.п. Такого рода условия возможно учитывать с помощью контекстно-зависимых грамматик, создание которых весьма нетривиально. Ещё более сложной представляется задача генерации конечных и детерминированных программ (динамически семантически правильных). Свойство детерминированности программы важно, так как позволяет утверждать наличие ошибки компилятора, если при его использовании наблюдается изменение поведения программы.

Ввиду сложностей создания контекстно-зависимых грамматик, генераторы конечных и детерминированных программ, как правило, представляют собой некоторую монолитную программу на языке высокого уровня (например, языке Perl). Такой генератор обычно очень сложно модифицировать, и поэтому он рассматривается как черный ящик, пригодный для генерации определенного класса программ некоторого языка. Существует подход к генерации исполняемых программ, удовлетворяющих некоторому набору предварительно заданных шаблонов, задающих вариацию некоторых параметров, однако гибкость такого подхода весьма ограничена и практически равноценна ручной разработке тестов.

Ввиду недостатков указанных подходов, для генерации тестов был выбран подход, основанный на использовании так называемой параметрической контекстно-свободной грамматики [3]. Существуют работы по использованию самомодифицирующихся контекстно-свободных грамматик [4], однако они проигрывают в наглядности и простоте параметрической контекстно-свободной грамматике.

Статья устроена следующим образом. В разд. 1 приводится общее описание параметрической контекстно-свободной грамматики. В разд. 2 находится описание основных моментов, связанных с разработанным генератором и грамматикой. Более формальное описание синтаксиса грамматики располагается в разд. 3. Разд. 4 состоит из примера грамматики и сгенерированной по ней программы. Статья завершается разд. 5, в котором освещена практика применения генератора.

1. ОПИСАНИЕ ПАРАМЕТРИЧЕСКОЙ КОНТЕКСТНО-СВОБОДНОЙ ГРАММАТИКИ

Каждый нетерминал параметрической контекстно-свободной грамматики имеет параметр, являющийся цепочкой, задающей контекст. Левая часть правил, помимо нетерминала, содержит некоторый образец, которому должен удовлетворять контекст нетерминала при определении возможности применить данное правило. Параметрическая контекстно-свободная грамматика по наглядности сопоставима с обычной контекстно-свободной грамматикой, но позволяет генерировать более широкой класс контекстно-зависимых языков. Например, в данной грамматике описывается генерация блочных операторов языка Pascal, в которых объявляются целочисленные и вещественные переменные и осуществляется присвоение между переменными только одного типа:

```
1) <S> ::= <PROG>
2) <PROG> ::= <BLK>IDLIST
3) <BLK>IDLIST = BEGIN <DCL>IDLIST <STM>IDLIST END
4) <DCL>TYPE ID,IDLISTV ::= ID:TYPE; <DCL>IDLISTV
5) <DCL>::=
6) <STM>IDLISTV1 TYPE ID, IDLISTV2 ::=
     ID := <VAR>TYPE*IDLISTV1 TYPE ID, IDLISTV2;
     <STM>IDLISTV1 TYPE ID,IDLISTV2
7) <STM>IDLIST1 ::= <BLK>IDLIST ; <STM>IDLIST1
8) <STM>IDLISTV ::=
```

```
 9) <VAR>TYPE*IDLISTV1 TYPE ID, IDLISTV2 ::= ID
10) <TYPE> ::= INTEGER | REAL
```

Идентификатор (имя переменной) задаётся извне нетерминалом ID. Под нетерминалом IDLISTV (задаваемым специальным образом) подразумевается список уникальных типизированных идентификаторов, включая пустую цепочку[1]. В правилах 2 и 7 непустой список IDLISTV, обозначаемый как IDLIST, находится в правой части правил, не встречается в их левых частях, и поэтому задаёт новый случайный список. В правиле 9 повторение слова TYPE задаёт ту же цепочку, что и первое использование этого слова. Правило 9 отвечает за выбор переменной с типом, равным типу переменной, стоящей в левой части присваивания правила 6.

При генерации допустимые правила выбираются случайным образом, что не даёт гарантий достижимости всех правил. Как показывают последние исследования в этой области, такая ситуация характерна и для других современных подходов к тестированию статической и динамической семантики компиляторов [2]. Например, язык, описываемый приведённой выше грамматикой, содержит следующую цепочку[2]:

```
BEGIN
  T : INTEGER;
  G : REAL;
  A : REAL;
  C : REAL;
  G := A;
  BEGIN
    B : REAL;
    G : INTEGER;
    A : REAL;
  END;
  T := T;
END
```

Для получения цепочки, соответствующей читаемой программе с переносами строк и соответствующими отступами, дополнительно предполагается использовать самоочевидные команды $RIGHT, $LEFT и $LINE:

```
<DCL>TYPE ID, IDLISTV ::= ID:TYPE; $LINE <DCL>IDLISTV
```

[1] Например, цепочка «integer T,real G,real A,» принадлежит языку IDLISTV.

[2] При использовании правил 2 и 7 соответственно создаются списки IDLIST «integer T,real G,real A,real C,» и «real B,integer G,real A,».

2. ОБЩЕЕ ОПИСАНИЕ ГЕНЕРАТОРА И ГРАММАТИКИ

Для создания генератора параметрической контекстно-свободной грамматики требовалось разрешить несколько неясностей оригинальной статьи [3]. Первая неясность связана с алгоритмом определения кратчайших правил грамматики, необходимых алгоритму генерации для возможности гарантированного завершения генерации и, тем не менее, порождения цепочки, принадлежащей языку грамматики. Пояснения к алгоритму были опубликованы позже в другой работе [5].

Вторая неясность связана с распознаванием образца левой части правил. В оригинальной статье для этого предлагается использовать нетерминалы этой же грамматики. Такой подход может оказаться крайне неэффективным, так как даже если используемый для этого нетерминал задаёт контекстно-свободный язык, не определяемый через параметризированные правила, то в общем случае его распознавание возможно за кубическое время от длины распознаваемой цепочки.

В разработанном генераторе распознавание осуществляется путём использования обычных регулярных выражений языка Python. Регулярные выражения допускают распознавание за линейное время и достаточны для распознавания контекста, достаточного для генерации программ реальных языков программирования. Язык Python был выбран в качестве языка реализации генератора[3] по причине того, что в его регулярные выражения включена возможность давать имена распознанным группам символов[4]. Например, здесь распознается и используется в правой части правила имя группы vars:

```
assign[.*@(?P<vars>.+)] = lref[vars] " = " int_expr[*] ";"
```

Распознавание контекста может состоять из нескольких регулярных выражений, причём в последующих выражениях можно использовать имена групп символов, распознанных ранее. Также для выборки случайного элемента списка (полученного разбиением строки некоторым разделителем) существует специальный синтаксис[5]. Например, следующее правило сначала обозначает весь непустой контекст именем vars, разделяет строку vars на

[3] Текст программы генератора занимает около 1000 строк хорошо комментированного кода.

[4] В распространённых версиях языка Perl, распознанные группы символов можно использовать только по их порядковому номеру.

[5] Выбор совпадения случайным образом (из всех возможных совпадений) нереализуем с помощью обычных регулярных выражений.

элементы символом запятой и случайным выбирает элемент списка, удовлетворяющий второму регулярному выражению в квадратных скобках:

```
lref [(?P<vars>.+)] vars=~?",":[\w+ (?P<id>\w+)] = id
```

В генераторе реализовано несколько интересных идей, возможно, использующихся в контексте генерации тестов впервые. Например, от языка Python была позаимствована идея использовать отступы в многострочных правилах грамматики для генерации отступов (и соответствующих переводов строк) в генерируемой цепочке символов. Благодаря этому, описание грамматики языка программирования и порождаемые ею программы становятся более наглядными без дополнительных пометок в грамматике. Например, следующее правило генерирует блочный оператор, ограниченный фигурными скобками, содержимое которых смещено вправо:

```
block [(?P<ftype>\w+)@(?P<old_vars>.*);(?P<new_vars>.*)] =
  "{"
    decls[new_vars]
    stmts[ftype "@" set_join[new_vars ";" old_vars]]
  "}"
```

Генератор поддерживает несколько правил специального вида для облегчения генерации программ языков программирования. Во-первых, существует правило для генерации цепочки символов, состоящей из уникальных частей, разделенных некоторым разделителем. Данное правило полезно для порождения уникальных последовательностей идентификаторов. Например, следующее правило порождает от нуля до трёх уникальных идентификаторов, разделённых символом запятой:

```
ids = { sep=",", min=0, max=3 } id
```

Во-вторых, существуют правила, которые для некоторого нетерминала запрещают генерацию цепочек, удовлетворяющих некоторому регулярному выражению. Данное правило полезно для генерации идентификаторов, не принадлежащих множеству ключевых слов языка программирования. Например, следующая последовательность правил генерирует цепочку букв, которая не может быть равна слову «for»:

```
id  = letter letters
id -= "for"
```

Как в первом, так и во втором случае может происходить повторная генерация нетерминала[6]. Количество попыток ограничено некоторой (достаточно большой) константой. При исчерпании доступного количества попыток грамматика считается неправильной и генерация выходной цепочки аварийно прерывается. Если для текущего нетерминала и его контекста не существует правила, им удовлетворяющего, то также происходит аварийное прекращение генерации цепочки, выводится стек нетерминалов и их контекстов, приведших к аварийной ситуации.

В грамматике есть средства для дополнительного управления процессом генерации выходной цепочки. Во-первых, для каждого правила существует возможность указать его вес, который прямо пропорционально влияет на вероятность срабатывания данного правила среди всех применимых правил[7]. Например, данное правило будет порождать в три раза (18 / 6 = 3) больше операторов присваивания (assign), чем условных операторов (if):

```
stmt = *18 assign[*] | *6 if[*]
```

Во-вторых, для каждого правила можно вручную указать кратчайшую альтернативу[8], когда алгоритм автоматического её определения не может сделать этого ввиду потенциальной бесконечности всех альтернатив. Если среди всех применимых альтернатив кратчайшая альтернатива неизвестна, то происходит аварийное прекращение генерации цепочки и выводится стек нетерминалов и их контекстов, приведших к аварийной ситуации. Например, в данном правиле указывается («!1»), что первая альтернатива («int_expr2[*]») обычно самая короткая (и генерируется в два раза чаще, чем вторая альтернатива):

```
int_expr = !1 *2 int_expr2[*]
         | int_expr[*] " " op2 " " int_expr[*]
```

3. ФОРМАЛЬНОЕ ОПИСАНИЕ ГРАММАТИКИ

Грамматика состоит из строк, каждая из которых может заканчиваться строковым комментарием, начинающимся с символа «#» (для упрощения

[6] Повторная генерация может быть вызвана нарушением свойства уникальности цепочки или генерацией запрещенной последовательности символов.

[7] Правильная балансировка правил важна для предупреждения излишней генерации одного и того же правила.

[8] Кратчайшая альтернатива используется при необходимости корректно завершить генерацию выходной цепочки (обычно при достижении ею определенного размера).

далее описаны только комментарии, занимающие целую строку и не являющиеся частью правил):

```
<грамматика> ::= <строки>
<строки> ::= <строка> [«\n»[9] <строки>][10]
<строка> ::= <правило> |[11] <комментарий> | <пустая строка>
  <комментарий> ::= # <произвольные символы>
```

Правило может быть как генерирующим (левая часть правила отделяется от правой части знаком «=»), так и ограничивающим (левая часть правила отделяется от правой части знаком «-=»):

```
<правило> ::= <левая часть правила> = <правая часть правила> |
  <левая часть правила> -= "<регулярное выражение Python>"[12]
```

Имя параметра шаблона контекста в левой части правила может быть любым именем распознанной группы символов регулярного выражения языка Python, стоящей в предыдущем шаблоне контекста левой части правила. Если имя параметра шаблона оканчивается символами «!~», то шаблон становится негативным (в случае наличия нетерминала разделителя ни один из элементов полученного списка не должен удовлетворять шаблону)[13].

```
<левая часть правила> ::= <имя нетерминала> [<шаблоны контекстов>]
<имя нетерминала> ::= <идентификатор>
<шаблоны контекстов> ::= <шаблон контекста> [<шаблоны контекстов>]
<шаблон контекста> ::= [<параметр шаблона>] [<разделитель>] <шаблон>
<параметр шаблона> ::= <имя параметра> =~ | <имя параметра> !~
<имя параметра> ::= <идентификатор>
<разделитель> ::= ?"<строка>"
<шаблон> ::= «[» <регулярное выражение Python> «]»
```

Правая часть правила может задаваться либо альтернативами, разделёнными вертикальной чертой, которые могут располагаться на нескольких строках, либо многострочным правилом, которое начинается с перевода строки (сразу после знака «=») и заканчивается пустой строкой. Отступы и переводы строк в многострочном правиле используются при его генерации.

---

[9] Здесь и далее в угловых двойных кавычках находятся строки нетерминалов, которые могут быть неоднозначно истолкованы. Под нетерминалом «\n» подразумевается перевод строки.

[10] В квадратных скобках находятся необязательные части грамматики.

[11] Прямая линия обозначает несколько возможных альтернатив правила.

[12] Прямая двойная кавычка обозначает нетерминал двойной кавычки.

[13] В негативном шаблоне нет смысла именовать распознанные группы.

```
<правая часть правила> ::= <альтернативы> | <многострочное правило>
<альтернативы> ::= <альтернатива> [ «|» [«\n»] <альтернативы>]
<альтернатива> ::= <модификаторы> <элементы> | <модификаторы> 0[14]
<многострочное правило> ::= «\n» <модификаторы>
                            <строки правила> <пустая строка>
<строки правила> ::= <строка правила> [«\n» <строка правила>]
<строка правила> ::= <элементы>
```

Все модификаторы необязательны. По умолчанию приоритет определяется автоматически, вес полагается равным единице, а в генераторе множества разделительная строка полагается равной пробелу, минимальное количество – единице, а максимальное количество – максимуму из значений min+19 и min*2.

```
<модификаторы> ::= [<приоритет>] [<вес>] [<генератор мн-ва>]
<приоритет> ::= ! <число>
<вес> ::= * <число>
<генератор мн-ва> ::= «{» [<параметры мн-ва>] «}»
<параметры мн-ва> ::= sep="<строка>"[, min=<число>][, max=<число>] |
                      min=<число>[, max=<число>] | max=<число>
```

Звездочка в качестве элемента правой части правила обозначает весь контекст нетерминала в левой части правила.

```
<элементы> ::= <элемент> [« » <элементы>]
<элемент> ::= "<строка терминалов>" | «*» | <имя параметра>[15] |
              <имя нетерминала> [«[» <элементы> «]»]
```

4. ПРИМЕР ГРАММАТИКИ И СГЕНЕРИРОВАННОЙ ПРОГРАММЫ

Далее приводится пример полной грамматики, отдельные правила которой уже рассматривались выше. Грамматика генерирует только правильные (компилируемые) программы на языке Си, имеющие детерминированное исполнение и завершающиеся за конечное время. Программы содержат скаляры и массивы целого типа, произвольные выражения целого типа от скалярных переменных и элементов массивов, операции присваивания, условные и блочные операторы.

[14] Для повышения наглядности и уменьшения ошибок пустое множество явным образом обозначается символом нуля.

[15] Имена распознанных групп (имена параметров) перекрывают имена нетерминалов.

```
#$ GEN_LENGTH = 512[16]

S = program[vars]
program = decls[*]
          "int main()"
          block["int" "@" * ";" vars]

### объявления и определения

decls[] = 0
decls[(?P<type>\w+) (?P<id>\w+)(?:,(?P<tail>.*))?] =
        type " " id " = " num ";"
        decls[tail]

decls[(?P<type>\w+) (?P<id>\w+)\[\],?(?P<tail>.*)] =
        type " " id "[4] = {" num ", " num ", " num ", " num "};"
        decls[tail]

### утверждения

stmts  = *3 stmts2[*] | stmts2[*] return[*]
stmt   = *18 assign[*] | *6 if[*]
stmts2 = 0
stmts2 = *3 stmt[*]
         stmts2[*]

assign[.*@(?P<vars>.+)] = lref[vars] " = " int_expr[*] ";"
assign[.*@] = ";"

if = if_part[*] | if_part[*] else_part[*]
if_part =
    "if ( " rel_expr[*] " )"
    block[* ";" vars]

else_part =
    " else"
    block[* ";" vars]

return [(?P<ftype>\w+)@.*] = "return " int_expr ";"

block [(?P<ftype>\w+)@(?P<old_vars>.*);(?P<new_vars>.*)] =
  "{"
    decls[new_vars]
    stmts[ftype "@" set_join[new_vars ";" old_vars]]
  "}"

### выражения

int_expr  = !1 *2 int_expr2[*] | int_expr[*] " " op2 " " int_expr[*]
```

[16] Генератор поддерживает некоторые прагмы. Например, прагма GEN_LENGTH задаёт количество символов в генерируемой последовательности, после которого генератор переходит в режим выбора кратчайших правил.

```
int_expr2 = !1 *10 term[*] | *2 "(" int_expr[*] ")" | "-" term[*]
rel_expr  = !1 *2 rel_expr2[*] |
              rel_expr[*] " " log_op2 " " rel_expr[*]
rel_expr2 = !1 *24 term[*] |
              *8  int_expr[*] " " rel_op2 " " int_expr[*] |
              *3  "!" term[*] | *1 "(" rel_expr[*] ")"
term                = num
term[.*@(?P<vars>.+)] = *2 lref[vars]
lref [(?P<vars>.+)] vars=~?",":[\w+ (?P<id>\w+)] = id
lref [(?P<vars>.+)] vars=~?",":[\w+ (?P<id>\w+)\[\]]
  = id "[" digit "]" | id "[(" int_expr[*] ")%4]"
op2     = "+" | "-"
rel_op2 = "<" | ">" | "=="
log_op2 = "||" | "&&"

### операции над множествами

ids        = { sep=",", min=0, max=3 } id
vars       = vars_gen[ids]
vars_gen[] = 0
vars_gen[(?P<id>\w+)(?P<sep>,?)(?P<tail>.*)]
  = type " " id sep vars_gen[tail] |
    type " " id "[]" sep vars_gen[tail]

# set_join бёрёт на вход два мн-ва vars A и B,
# разделённых «;» и возвращает мн-во vars = A U (B\A),
# причём B\A задаётся с помощью set_sub

set_join [(?P<s1>[^;]+);[^;]*] = s1 "," set_sub[*]
set_join [;(?P<s2>[^;]*)] = s2
set_sub [(?P<s1>.+);] = 0
set_sub [(?P<s1>.+);(?P<t>\w+)
(?P<id>\w+)(?P<arr>\[\])?(?P<sep>,?)(?P<s2>.*)]
        s1!~[(?:.+,)?\w+ (?P=id)(?:\[\])?(?:,.+)?] =
  t " " id arr sep set_sub[s1 ";" s2]

set_sub [(?P<s1>.+);\w+ (?P<id>\w+)(?:\[\])?,?(?P<s2>.*)]
        s1=~[(?:.+,)?\w+ (?P=id)(?:\[\])?(?:,.+)?] =
  set_sub[s1 ";" s2]

### идентификаторы, числа и типы (очень упрощенный вариант)

id      = letter letters
id     -= "for"
letter  = "f" | "o" | "r"
letters = letter letters | *10 0
num    = digit digits
digit  = "0" | "1" | "2" | "3"
digits = digit digits | *10 0
type = "int"
```

Приведённая выше грамматика порождает, например, следующую цепочку символов, являющуюся правильной программой на языке Си:

```
int f[4] = {3, 1, 0, 2};
int o = 0;

int main()
{

  if ( o < 2 + (-3) )
  {
    int o[4] = {2, 2, 0, 0};
    int of = 0;

    of = o[(01 - 33 - (3))%4];
    if ( 2 )
    {
      int rf = 0;

      f[(1)%4] = o[3];
      return 3;
    }
    of = (of);

  } else
  {
    int f = 30;


  }
  if ( f[(3)%4] )
  {

    return -2 + 1 + 0 - 0;
  } else
  {
    int o = 3;

    f[0] = 2 + o;
    return 2;
  }
  o = 1;
  f[3] = o;
  return 0 - (1) + (3);
}
```

5. ПРАКТИКА ПРИМЕНЕНИЯ ГЕНЕРАТОРА

В качестве доказательства полезности предлагаемого подхода к генерации тестов была написана простейшая грамматика для порождения детер-

минированных, исполняемых программ на языке Си-99. Грамматика нацелена на выявление ошибок во внутренних преобразованиях компилятора, и поэтому не содержит замысловатых синтаксических конструкций.

Грамматика генерирует программы с одной функцией (main), переменными и одномерными массивами разнообразных целочисленных типов. Функция состоит из случайной последовательности утверждений. Утверждением может быть присваивание переменной произвольному выражению, условный оператор (if-else) с произвольными логическими операциями, цикл for или while. Условные операторы и циклы содержат блочный оператор, также состоящий из случайной последовательности утверждений. Утверждением также может быть оператор вывода значения производной переменной или элемента массива и вызов специальной функции из внешнего модуля (недоступного компилятору), которая изменяет значение, переданное по его указателю (для проверки возможности этого изменения компилятором).

Индексные выражения для выбора элементов массива никогда не выходят за границы массива (для обеспечения конечности и детерминированности генерируемых программ) благодаря статическому или динамическому контролю. Статический контроль обеспечивается индексами, являющимися циклическими переменными с некоторым возможным фиксированным смещением. В свою очередь, количество итераций циклов контролируется статически или динамически и является инвариантом цикла. Грамматика также запрещает генерацию программ с произвольной модификацией циклических переменных (также во избежание появления бесконечных циклов).

Указанная грамматика занимает всего около 200 строк и описывает генерацию указанного класса программ вплоть до генерации отдельных символов. Генератор и указанная грамматика были опробована на бета версии 11.1 компилятора фирмы Intel® для операционной системы Linux. Программа компилировалась с оптимизациями и без них и, если вывод программы отличался, то производилась автоматическая минимизация программы (на текущий момент простейший алгоритм удаления строк) до программы, всё ещё дающей отличие. Минимизированная программа использовалась для дальнейшего анализа проблемы. За месяц работы генератора среди около полумиллиона тестов было найдено около двух десятков новых (разных) проблем в компиляторе, которые были признаны разработчиками компилятора и будут исправлены ими. Большинство найденных проблем вызвано ошибками в оптимизирующих преобразованиях, тогда как остальные проблемы вызваны аварийной остановкой компиляции.

ЗАКЛЮЧЕНИЕ

Разработан генератор, основанный на формализме параметрической контекстно-свободной грамматики и допускающий генерацию контекстно-зависимых языков, достаточных для автоматического построения семантически правильных (компилируемых) программ, имеющих детерминированное исполнение. Путём анализа различий вывода полученных детерминированных программ, скомпилированных с оптимизациями и без них, были обнаружены ошибки в бета версии компилятора промышленного уровня.

В дальнейшем планируется расширение грамматики для покрытия более широкого класса программ, включая возможности языка Си++, такие как обработка исключений. Есть идея усовершенствовать синтаксис грамматики путём добавления списков контекстов и операций для работы с ними вместо одной монолитной строки контекста. Логичным выглядит добавление возможности минимизации программы, диагностирующей ошибку компилятора, на основании дерева генерации[17].

СПИСОК ЛИТЕРАТУРЫ

1. Aho A.V., Sethi R., Ullman J.D. Compilers: principles, techniques, and tools. – Boston: Addison-Wesley Longman Publishing Co., Inc., 1986. – 796 p.
2. Kossatchev A. S., Posypkin M. A. Survey of compiler testing methods // Programming and Computing Software. – NY: Plenum Press, 2005. – Vol. 31, No. 1. – P. 10–19.
3. Bazzichi F, Spadafora I. An automatic generator for compiler testing // IEEE transactions on Software Engineering. – NY: IEEE, 1982. – Vol. SE-8. – P. 343–353.
4. Hanford K.V. Automatic generation of test cases // IBM Systems Journal – NY: IBM, Dec. 1970. – Vol. 9. – P. 242–257, Dec. 1970.
5. Malloy B.A., Power J. F. An Interpretation of Purdom's Algorithm for Automatic Generation of Test Cases // Proceedings of the First Annual International Conference on Computer and Information Science. – Orlando, Florida, USA, October 3-5, 2001. – P. 310–317.

[17] В корне дерева генерации находится начальный символ грамматики с пустым контекстом. В вершинах дерева генерации находятся нетерминалы и их контексты. Минимизация проводится путём попыток повторной генерации деревьев с корнем в некотором нетерминале в соответствии с его кратчайшим правилом.